DigMe: Digital Microelectronics

Naoki Pross – naoki.pross@ost.ch

Fall Semester 2021

Contents

1 License

2 Design Flow

3 Design constraints and static timing analysis (STA)

Synthesis and implementation tools can reduce VHDL code into a set of combinatoric and sequetial logic parts, but for the last step information about the hardware is required. Such information is given through the *constraints* defined through XDC[1] or SDC[2] files. Both file formats are mostly a set ot TCL commands.

Constraints should be generally organized in three sections (or separate files):

- Physical constraints: described below, usually before timing.
- Timing assertions: primary clocks, virtual clocks, generated clocks, clock groups, input and output delay constraints.
- Timing exceptions: false paths, min / max delay, multicycle paths, case analysis, disable timing.

3.1 Physical constraints

Physical contraints include: I/O contraints, Netlist constraints, Placement constraints, Routing constraings. Physical constraints are usually given through the graphical user interface.

3.2 Timing constraints

$$t_{\text{slack}} = T - t_{\text{arrival}}$$

4 System level VHDL

4.1 Aliases

The goal is now to build re-usable IP blocks with VHDL. For that we need to refresh some important features of the lanugage. The first of which are aliases.

```
signal data_bus:
  std_logic_vector(31 downto 0);
alias first_nibble:
  std_logic_vector(0 downto 3)
    is data_bus(31 downto 28);
```

4.2 Generics

4.3 Generators

Another useful feature are `generate` statement, with the syntax that allows the instantiation of multiple components.

[1]Xilix Design Constraints, proprietary format.
[2]Synopsys Design Contraints, industry standard.

```
[label]: for ⟨identifier⟩ in ⟨range⟩ generate
  -- optional declaration part
  -- begin only required if there is a
    ↪ declaration
[begin]
  -- concurrent statements
end generate [label];
```

For example:

```
for i in 0 to 7 generate
  x(i) <= a(i) xor b(7 - i);
end generate;
```

Or in a more realistic case, with components imported from elsewhere.

Listing 1: Example of generate with a component.

```
-- in architecture
bcd_to_sseg_inst_loop:
for i in 0 to nr_digits - 1 generate
  bcd_to_sseg_inst: component bcd_to_sseg
    port map(
      clk => clk,
      rst => rst,
      bcd => bcd_array(i),
      sseg => sseg_array(i)
    );
end generate;
```

where `bcd_array` and `sseg_array` are of course array types, and `nr_digits` is a constant.

4.4 Functions and procedures

Furthermore VHDL has functions that can be useful to avoid rewriting the same code. Function have multiple inputs and a signel output, are allowed to be called recursively, but cannot declare or assign signals, nor use `wait` statements.

```
function ⟨name⟩ ([list of arguments with type])
  return ⟨return type⟩
is
  [declaration of variables]
begin
  -- sequential statement (but not wait)
end function ⟨name⟩;
```

An example is a parity generator:

Listing 2: An odd parity generator function.

```
function pargen(avect: std_ulogic_vector)
  return std_ulogic
is
  variable po_var : std_logic;
begin
  po_var := '1';
  for i in avect'range loop
    if avect(i) = '1' then
      po_var := not po_var;
    end if;
  end loop;
  return po_var;
end function pargen;
```

In testbenches it is common to see procedures. They differ for function as they can have multiple inputs and *multiple output*. Because of this they in practice are usually not synthetizable. The syntax is similar to functions:

```
procedure ⟨name⟩ ([list of arguments with
    direction]) is
  [declaration of variables]
begin
  -- sequential statement
end procedure ⟨name⟩;
```

With *list of arguments with direction* it is meant an expression like `a, b : in real; w : out real`, similar to the arguments of `port`.

4.5 Arrays and records

To efficiently use `generate` statement, such as in listing 1, we ned array types. Arrays types (fields) of other types are defined with the following syntax.

```
type ⟨name⟩ is array (⟨upper
    limit⟩ downto ⟨lower limit⟩) of ⟨base type⟩;
```

For example to complete listing 1, we create 1 by 1 matrices.

```
constant nr_digits : integer := 3;
type bcd_array_type is
  array (0 to nr_digits -1)
  of std_ulogic_vector(3 downto 0);
type bcd_array_type is
  array (0 to nr_digits -1)
  of std_ulogic_vector(6 downto 0);
```

While all arrays elements must have the same underlying type, *records* allow for different types to be combined together. For example:

```
type memory_access is record
  address : integer range 0 to
    ↪ address_max -1;
  mem_block : integer range 0 to 3;
  data : std_ulogic_vector(word_width -1
    ↪ downto 0);
end record;
```

4.6 Packages

To declare your own packges, the syntax is rather easy:

```
⟨library and / or use statements⟩
package ⟨name⟩ is
  [declarations]
end package ⟨package name⟩;
```

And possibly in another file the implementation is give with:

```
package body ⟨name⟩ is
  ⟨list of definitions⟩
end package body ⟨name⟩;
```

In practice it is common to see for example a configuration package, that contains all constants for the project. For example if we were to put the function `pargen` from listing 2 we could do:

```
package parity_helpers is
  constant nibble : integer;
  constant word   : integer;
  function pargen(avect :
      ↪ std_ulogic_vector) return
      ↪ std_ulogic;
end package parity_helpers;

package body parity_helpers is
  -- functions
  function pargen(avect:
      ↪ std_ulogic_vector)
    return std_ulogic
  is From listing 2
  end function pargen;
  -- instantiation of variables
  constant nibble : integer := 4;
  constant word   : integer := 8;
end package body parity_helpers;
```

And later use it with `use work.parity_helpers.all`.

4.7 Fixed point arithmetic

4.7.1 Mathematics

Recall that a binary number is represented using weights 'bits' $a_k \in \{0,1\}$ in front of powers of 2, so that an integer $z \in \mathbb{N}_0$ is represented with n bits as

$$z = \sum_{k=0}^{n-1} a_k 2^k.$$

Thus with n bits we can represent the integer range $\{0, \dots, 2^{k-1}\}$. To expand this to negative integers we shift everything by 2^k obtaining

$$z = -(2^{n-1})a_{n-1} + \sum_{k=0}^{n-2} a_k 2^k,$$

which allows for values in the asymmetric integer range $\{-2^{n-1}, \dots, 2^{n-1} - 1\}$. To further extend this to the rational numbers we add to the n integer bits, m fractional bits, such that

$$z = \sum_{k=-m}^{n-1} a_k 2^k.$$

This format is known as the `Qn.m` or $Q(n,m)$ format. It is however not specified if the values are to be interpreted as signed or unsigned. For that there are: `uQn.m` for unsigned values which has range 0 to $2^n - 2^{-m}$, and `sQn.m` for signed values with range -2^{n-1} to $2^{n-1} - 2^{-m}$ (same as previous but shifted by 2^n).

When doing calculations using `Qn.m` with different sizes, generally the following rules apply:

$$\begin{aligned} &Q(n_1,m_1) + Q(n_2,m_2) \\ &\qquad = Q(\max(n_1,n_2)+1, \max(m_1,m_2)) \\ &Q(n_1,m_1)\cdot Q(n_2,m_2) = Q(n_1+n_2, m_1+m_2) \end{aligned}$$

There is an edge case for products between `sQn.m` numbers where we can save a bit using

$$\begin{aligned} &Q_s(n_1,m_1)\cdot Q_s(n_2,m_2) \\ &\qquad = Q_s(n_1+n_2-1, m_1+m_2+1) \end{aligned}$$

4.7.2 Manual implementation

To manually implement fixed point arithmetic in VHDL, we can use `integer` types by left shifting the numbers by the right amount. For example:

```
constant A : real = 2.5248;
-- to convert this into a uQ2.6 (8 bits)
-- we have to left shift by 2^6.
variable a_fix : unsigned(7 downto 0) :=
  to_unsigned(integer(2.0 ** 6 * A), 8);
-- Note that by keeping only 6 digits
-- the value is truncated down to 2.5156
```

4.7.3 With VHDL 2008 `fixed_pkg`

In VHDL 2008 there is a fixed point arithmetic library, which is unfortunately not completely standardized yet. It it not always optimum in terms of resource usage and speed because it guarantees overflow prevention. To import it we simply add the following:

```
-- since VHDL 2008
library ieee;
use ieee.fixed_generic_pkg.all;
use ieee.fixed_float_types.all;
```

```
-- older versions
library ieee_proposed;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.fixed_float_types.all;
```

To represent `uQn.m` numbers the syntax is:

```
signal f : ufixed(⟨n⟩ downto -⟨m⟩);
```

The minus sign implies that the values are after the comma. To write `sQn.m` numbers there is the type `sfixed`. Actually any range is valid, so it is possible to write:

```
signal f : ufixed(-2 downto -3);
```

Conversion operators `to_ufixed`, `to_sfixed` are available. To get back n and m the attributes `'high` resp. `'low` are available. Furthermore the library allows to control the behaviour of conversions and range limits, through arguments of `to_ufixed` (or `to_sfixed`).

- Overflow is controlled with `overflow_style` set to `fixed_saturated` (default, value cannot increase / decrease) or `fixed_wrap` (over / underflow).
- Rounding can be controlled by setting `roud_style` to `fixed_round` (default) or `fixed_truncate`.